Transforming Software Testing with Function Extraction

Alan R. Hevner
University of South Florida and
CERT STAR*Lab
Software Engineering Institute
Carnegie Mellon University
achevner@coba.usf.edu

Richard C. Linger and Mark G. Pleszko
CERT STAR*Lab
Software Engineering Institute
Carnegie Mellon University
rlinger@sei.cmu.edu
mpleszko@cert.org

Abstract

The craft of software testing can benefit from technologies that enable evolution toward a true engineering discipline. In current practice, software developers lack practical means to determine the full functional behavior of programs under development, and even the most thorough testing can provide only partial knowledge of behaviors. Thus, an effective technology for revealing software behaviors could have a positive impact on software testing. This paper describes the emerging technology of function extraction (FX) for computing the full behavior of programs to the maximum extent possible with mathematical precision. We explore how the use of FX technologies can transform methods for functional verification of software. An example illustrates the value of full behavior knowledge for complete and confident assessment of software quality and fitness for use. We conclude by discussing FX impacts on the field of software testing.

2. Function Extraction Concepts

CERT STAR*Lab of the Software Engineering Institute at Carnegie Mellon University is conducting research and development in the emerging technology of function extraction. The objective is to compute the behavior of software to the maximum extent possible with mathematical precision. FX presents an opportunity to reduce dependencies on slow and costly testing processes to assess software functionality by moving to fast and cheap computation of functionality at machine speeds. Because a principal objective of testing is to validate functionality, automated computation of functional behavior can be expected to streamline testing processes and permit increased testing focus on system-level issues of component interaction and performance.

The goals of behavior calculation are to compose and record the semantic information in programs as a means to augment human capabilities for analysis, design, and verification. We discuss function extrac-
tion below in the context of sequential logic, in full knowledge that, say, concurrent and recursive logic must be addressed as well. Computing the behavior of programs is a difficult problem, and our intent is to say first words on the subject, not last words.

The well-known function-theoretic view of software provides mathematical foundations for computation of behavior [Linger et al. 1979, Pleszkoch et al. 1990, Prowell et al. 1999]. In this perspective, programs are treated as rules for mathematical functions or relations, that is, mappings from inputs (domains) to outputs (ranges), regardless of subject matter addressed or implementation languages employed.

The key to the function-theoretic approach is the recognition that, while programs may contain far too many execution paths for humans to understand or computers to analyze, every program (and thus every system of programs) can be described as a composition of a finite number of control structures, each of which implements a mathematical function or relation in the transformation of its inputs into outputs. In particular, the sequential logic of programs can be expressed as a finite number of single-entry, single-exit control structures: sequence (composition), alternation (ifthenelse), and iteration (whiledo), with variants and extensions permitted but not necessary. The behavior of every control structure in a program can be extracted and composed with others in a stepwise process based on an algebra of functions that traverses the control structure hierarchy. Termination of the function extraction and composition processes are assured by the finite number of control structures present in any program [Linger and Pleszkoch 2004].

The first step in behavior extraction is to transform any spaghetti logic in the input program into structured form, to create a hierarchy of nested and sequenced control structures. The behavior of leaf node control structures is then computed with net effects propagated to the next level while local details of processing and data are left behind. These computations reveal new leaf nodes and the process repeats until all behavior has been computed.

Behavior computation for sequence and alteration structures involves composition and case analysis. Because no comprehensive theory for loop behavior computation can exist, mathematical foundations and engineering implementations short of a general theory but sufficient for practical use are under development.

The general form of the expressions produced by function extraction is a set of conditional concurrent assignments (CCA) organized into catalogs that define program behavior in all circumstances of use. The CCAs are disjoint and thus partition behavior on the input domain of a program. The catalogs define behavior in non-procedural form and represent the as-built specification of a program. Each CCA is composed of a predicate on the input domain, which, if true, results in simultaneous assignment of all right-hand side domain values in the concurrent assignments to their left-hand side range variables.

Behavior catalogs, thus, are the central repository for the actual behaviors contained in a software system. The catalogs can be queried, for example, for particular behavior cases of interest, or to determine if any cases satisfy, or violate, specified conditions or constraints. Behavior catalogs have many uses, as seen in Figure 1, ranging from basic human understanding of code, to program correctness verification, to analysis of security and other attributes, to component composition, and so on [Hevner et al. 2005]. In the next section, we will focus on the specific impacts of the FX technology on software testing activities.

The first application of FX technology is to programs written in, or compiled into, Intel assembly language to support analysts in malicious code detection and understanding of malware behaviors. Sample outputs from the evolving FX system are employed later in the paper to illustrate the role of behavior computation as a means to satisfy testing objectives.

3. Software Testing with FX Technology

The impacts of FX technology resonate throughout the full range of software development activities [Hevner et al. 2005]. In this section, we focus on how FX can transform testing practices during software-intensive system development.

3.1 Coding and Unit Testing

When a decision is made to develop a required software component from scratch, FX automation can play an important role during the evolving implementation. As each set of required functions is developed, a software developer can work interactively with an FX system to determine if the evolving implementation indeed provides the set of functions intended. As new code is introduced into an evolving component, the FX system can report on the corresponding additional behaviors, as well as any changes to prior behaviors. Errors of commission or omission can thus be identified during the implementation process, and extraneous behavior isolated and removed.
3.2 Component Correctness Verification

Significant time and effort are often allocated during software development to verify the correctness and quality of software designs and implementations. Reviews, inspections, and unit testing are resource-intensive activities used to evaluate components against their specifications. At its core, FX technology is closely related to correctness verification. Programmers can add intended functions (expressed in a standard language form as comments) to the control structures of implementations to permit FX automation to compare the extracted behavior of each control structure to the corresponding intended function to determine whether or not it is correct. The addition of FX automated support to front-end verification activities can lead to more efficient and effective methods of designing and building software components.

3.3 Integration Testing

Function Extractors are essentially generalized composition engines, and they can also play a role in the integration of software components as determined by a system architecture. Based on the behavior catalog of each component, FX technology, guided by mathematical rules of component composition, can be adapted to integrate uses of the components into an assembled subsystem with a new, composite behavior catalog. The architecture specifies intended and allowable usage patterns (i.e., control flows and data flows) among the integrated components.

3.4 System Testing

With the advent of FX technology, an opportunity exists for subsystem, system, and customer acceptance testing to shift from defect detection to certification of fitness for use. Subsystems and entire systems could be processed by FX automation, and resulting behavior catalogs compared with specifications and analyzed by stakeholders. A reduced set of test scenarios can be developed to demonstrate correct execution, because only one test per disjoint case of behavior is sufficient to validate all the behavior defined by that case. Of course, testing of assumptions regarding environmental conditions, hardware platforms, and usage patterns must be carried out as well.

Additional testing effort can be devoted to validating the level of quality attributes provided by the system. For example, system testing for the qualities of performance, security, privacy, reliability, survivability, and maintainability, to mention a few, will become a greater focus of system testing [Walton et al. 2006].

Another important consideration is that eventual industry standards for FX technology could support
outsourcing of system testing to independent groups that specialize in certifying the correctness and quality of software systems. As in more mature engineering fields, independent certification of quality standards for software systems with an industry-wide stamp of approval will help provide greater levels of trust in critical systems.

3.5 System Maintenance and Evolution

It is generally accepted that approximately 80% of the cost of a software system occurs after it is deployed, in the form of maintenance and upgrades to meet evolving customer requirements. FX technology can support maintenance and evolution activities while providing opportunities for cost savings and quality improvements.

The key to system maintenance with FX technology is keeping behavior catalogs up to date automatically. As maintenance is performed on an operational system (for example, to improve performance or enhance security), the resulting system must still produce the same intended behaviors for unaffected functions as found in the catalog. As in system testing, a reduced set of regression test scenarios can provide a level of confidence that unaffected behaviors have remained unchanged.

In terms of system evolution, behavior catalogs provide a formal baseline against which all changes can be compared. New or modified behaviors can be specified initially in specification behavior catalogs and traced through component design and implementation catalogs. Thus, developers can determine where and how to make required changes in system specifications, component designs, and code. Once code changes are made, FX automation can help ensure they have desired effects, while checking the integrity of behaviors that must remain unmodified.

Even when an operational system is not subject to maintenance and evolution activities, it may be wise to periodically perform function extraction to help ensure that no malicious or inadvertent modifications have been introduced. Frequent application of the FX technology can help provide users with a level of confidence that no security compromises have occurred since the previous FX analysis.

3.6 Component Acquisition

Components that are acquired from external vendors or even from internal reuse repositories present major challenges to developers who must understand their behaviors. FX automation can provide a solution. A function extractor based on the semantics of the component’s programming language can accept an unknown component and produce a complete behavior catalog. The resulting behavior catalog can then be analyzed and compared to its component design catalog, as well as to system specification and architecture catalogs, if available. By evaluating several components in this manner, developers can create a basis for the best selection to meet acquisition requirements.

As examples of the application of FX technology for component evaluation and selection, consider the following situations:

- **Legacy programs** - A developer submits a legacy program to an FX system to understand its behavior in order to integrate it with newly developed components.
- **COTS products** - A systems engineer requests a product behavior catalog from a COTS vendor to evaluate its planned use in a new system.
- **Service integration** - Before signing an agreement to include an online service in a critical supply chain application, a systems integrator requires the service provider to run the service through an FX system in order to analyze the full set of service behaviors. Note that the provider need not expose any proprietary code to the service user, only the service behaviors.

4. A Behavior Computation Example

In miniature illustration of the role of FX in a testing environment, consider the problem of developing a program that returns 1 if three given integers form the sides of a proper triangle, otherwise it returns 0. The C language program depicted below appears to satisfy this requirement by first checking that each side is a positive number, and then by checking each permutation of the triangle inequality:

```c
int test_triangle (int a, int b, int c) {
    int answer = 1;
    if (a <= 0) {
        answer = 0;
    }
    if (b <= 0) {
        answer = 0;
    }
    if (c <= 0) {
        answer = 0;
    }
    if (a + b <= c) {
        answer = 0;
    }
    if (a + c <= b) {
        answer = 0;
    }
    if (b + c <= a) {
        answer = 0;
    }
    return answer;
}
```
Since the FX system computes behavior for compiled machine code, the first step is to disassemble the object code to produce the listing shown below. The IDA Pro disassembler is used for this purpose:

```c
; test_triangle(int, int, int)
public __Z13test_triangleiii
__Z13test_triangleiii proc near
var_4  = dword ptr -4
arg_0  = dword ptr  8
arg_4  = dword ptr  0Ch
arg_8  = dword ptr  10h
push ebp
mov ebp, esp
sub esp, 4
mov [ebp+var_4], 1
cmp [ebp+arg_0], 0
jg short loc_1A
mov [ebp+var_4], 0
loc_1A: cmp [ebp+arg_4], 0
jg short loc_27
mov [ebp+var_4], 0
loc_27: cmp [ebp+arg_8], 0
jg short loc_34
mov [ebp+var_4], 0
loc_34: mov eax, [ebp+arg_4]
add eax, [ebp+arg_0]
```  

Next, the FX system is executed from an IDA Pro plug-in, with a screen shot of the resulting output depicted below in Figure 2. On the left side of the screen, the spaghetti-logic of the disassembled C program has been transformed into structured form. On the right side, the computed behavior is presented in terms of the net effect of the program on registers, flags, and memory.

![Figure 2: Triangle Program in FX System](image-url)
There are two cases (conditions) in the computed behavior, each defined as a conditional concurrent assignment. The registers section of the first case shows that the EAX register is set to 0, resulting in returning 0 to the calling program. Additionally, the EBP register is used but is finally set back to its original value, and the ESP register is incremented by 4 (by the RET instruction which pops the return address off the stack). The memory section shows the final value of the local variable “answer” in the original C program, as well as other data that were saved on the stack. The second case is similar, except that the EAX register is set to 1, thereby returning 1 to the calling program. Thus, provided that the conditions themselves are correct, there is no need to execute any test cases on this program to determine its functional behavior.

The condition for the second case is as follows:

\[
\begin{align*}
&\text{parm}_a \leq (\text{signed}_32(\text{parm}_b + d \text{parm}_c) + -1) \\
&\text{&&} (\text{parm}_b \leq (\text{signed}_32(\text{parm}_a + d \text{parm}_c) + -1)) \\
&\text{&&} (\text{parm}_c \leq (\text{signed}_32(\text{parm}_a + d \text{parm}_b) + -1)) \\
&\text{&&} (1 \leq \text{parm}_a) \\
&\text{&&} (1 \leq \text{parm}_b) \\
&\text{&&} (1 \leq \text{parm}_c)
\end{align*}
\]

where

\[
\begin{align*}
\text{parm}_a &\,:= \text{signed}_32(\text{acc}_32(\text{M}, 4 + d \text{ESP})) \\
\text{parm}_b &\,:= \text{signed}_32(\text{acc}_32(\text{M}, 8 + d \text{ESP})) \\
\text{parm}_c &\,:= \text{signed}_32(\text{acc}_32(\text{M}, 12 + d \text{ESP}))
\end{align*}
\]

For readability, the output of FX/ASM has been manually formatted to use symbolic names for the function parameters instead of the low-level memory access expressions shown in the definitions of those parameters. The evolving system will soon be able to perform this formatting automatically.

The condition for the program to return 1 is the logical “and” of six checks, corresponding to the six ifthenelses in the program. The checks for positive parameters are easily seen to be correct. However, an examination of the other checks shows that the overflow case has not been considered. (Note that “+d” in the condition represents addition modulo 2^32, and the “signed_32” operation interprets the result as a two’s complement signed number.) As a result, the program gives the wrong answer whenever the addition of two of the sides causes an arithmetic overflow. For example, as presently implemented, the program would incorrectly indicate that 2^30, 2^30, and 10 do not form the sides of a triangle. Note that this error would likely not have been found by any of the common test coverage strategies, including statement coverage, branch coverage, and path coverage.

Consider next the problem of determining whether malicious content has been added to this program, in particular, whether the EAX register is being maliciously employed under conditions defined by an intruder.

The FX screenshot in Figure 3 shows computed behavior for such an instance. A third case of behavior appears, where the malicious intent is invoked only for inputs that cause the addition of sides b and c to overflow. The assignment of a value of 80 to EAX could cause a buffer overflow or some other problem in the calling program, which is expecting a return value of at most 1. This example illustrates the discovery and exploitation of a program vulnerability by an intruder. This malicious execution would be just as difficult to find through testing as the overflow error in the original program. However, the FX system immediately computes a third case of behavior with a return value of 80, revealing the malicious functionality for all to see.

5. Discussion of FX Impacts

It is not surprising that the software testing process can benefit immensely from a thorough analysis of the code being tested. Test coverage metrics, such as statement or branch coverage, are one example of this. There are also situations where examination of the code and its structure can be used to reduce the number of test cases needed without the possibility of missing an error in the program. For example, if both the code and the specification are linear functions (in the mathematical sense) of N numeric input values, then any N+1 linearly independent test cases suffice to demonstrate correctness. For functions of a single numeric input value, if the code and the specification are polynomials of degree M, then any M+1 distinct test cases suffice.

However, if all that is known about the code and the specification is that they are total recursive functions, then every possible input value must be tested.
The advent of function extraction technology provides the ability, outside the strict confines of linear or polynomial functions, to reduce the number of test cases without the possibility of missing a program error. In fact, the need for test cases can be eliminated altogether.

Observation 1: If nothing is known about the internals of a sequential program (a black box as far as testing is concerned), then in order to guarantee correctness, it is necessary to execute every possible test case where the program specification makes a non-trivial requirement about the program output.

Observation 2: For a sequential program, if the functional behavior extracted from the program implementation satisfies the program specification, then the program is functionally correct, and no testing for functional behavior is required.

These observations clearly identify the paradigm shift supported by FX capabilities for software testing. No longer are testing coverage metrics crucial to decisions about whether sufficient testing has been completed. All behaviors in the program code are identified and can be inspected and evaluated for correctness against the program specification.

Comprehensive testing efforts for large systems have many objectives to satisfy. One of these objectives is to gain confidence that the components comprising a system satisfy their specifications, whether the specifications are written down or exist only as mental models. The function extraction process produces catalogs of as-built program behaviors that can be inspected and queried by programmers and analyzed by machines to determine conformance, or not, to specifications. Functional testing can provide no additional information at whatever cost.

Function extraction may also make it easier to determine if malware or corrupted functionality is present in operational programs. Behavior catalogs can be generated on a periodic basis and compared with baseline catalogs to help detect any malicious content.

As noted, many other testing objectives must be satisfied, including evaluation of the performance and
interaction of programs in complex computational environments. Function extraction has the potential to free testing resources to focus on these objectives with the knowledge that the functional behavior of constituent programs is known and validated. This resource shift may impact the economics of software engineering, resulting in faster and cheaper development of higher quality systems [Collins et al. 2005].

6. References


